John von Neumann Institute for Computing

A Load Balancing Framework in Multithreaded Tomographic Reconstruction

José Antonio Álvarez, Javier Roca Piera,
José Jesús Fernández

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr,
F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 165-172, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

http://www.fz-juelich.de/nic-series/volume38

A Load Balancing Framework in Multithreaded Tomographic Reconstruction

José Antonio Álvarez, Javier Roca Piera, and José Jesús Fernández

Departamento de Arquitectura de Computadores y Electrónica
Universidad de Almería, 04120 Almería, Spain
E-mail: {jaberme, jroca, jose}@ace.ual.es

Clusters are, by now, one of most popular architectures. Hiding latencies as well as getting an optimal assignment of processors, are two issues for many scientific applications, specially if non dedicated clusters are used. Traditionally, high performance scientific applications are parallelized using MPI libraries. Typical implementations of MPI, minimize dynamic features required to face latencies or shared resource usage. Switching from the MPI process model to a threaded programming model in the parallel environment, can help to achieve efficient overlapping and provide abilities for load balancing. Gains achieved by $BICAV$ (our research group's tomographic reconstruction software) in a multithreaded framework, are exposed.

1 Introduction

Multithreaded programming[1] provides a way to divide a program into different separated pieces that can run concurrently. Using user level threads[2,3] in places where concurrence can be exploited, would allow us to achieve latency hiding, better scalability, skills to avoid overhead on processors due to faster context switching and, also, key abilities to migrate threads for load balancing purposes. In the design of a parallel and multithreaded strategy for applications related to image processing, such as 3D tomographic image reconstruction algorithms[4], data distributions have to be carefully devised so locality is preserved as much as possible. Specially in systems like non-dedicated clusters, where the workload needs to be dynamically reassigned. The strategy presented, RakeLB (inspired on Rake[5] algorithm), has been implemented following a centralized scheme and seizes the facilities provided by AMPI [6,7], which is the top level layer of the Charm [8] framework. AMPI embeds MPI processes into user level threads (*virtual processors* or *VPs* in Charm framework), allowing more than one active flow of control within a process, therefore aspects like context switching and migration of tasks are possible with a relative efficiency. Applications like $BICAV$ can run on non-dedicated clusters without significant performance loss, if latency hiding and adaptability are achieved and combined.

This work presents and analyzes results obtained by $BICAV$[9] when it was ported to a multithreaded environment. Section 2 describes $BICAV$, Section 3 exposes gains for the multithreaded version of $BICAV$ due to latency hiding. Section 4, describes the load balancing strategy, RakeLB, compared to more general load balancing strategies. Section 5 evaluates RakeLB and finally, conclusions are presented in Section 6.

2 Iterative Reconstruction Methods: BICAV

Series expansion reconstruction methods assume that the 3D object, or function f, can be approximated by a linear combination of a finite set of known and fixed basis functions,

with density x_j. The aim is to estimate the unknowns, x_j. These methods are based on an image formation model where the measurements depend linearly on the object in such a way that $y_i = \sum_{j=1}^{J} l_{i,j} \cdot x_j$, where y_i denotes the i^{th} measurement of f and $l_{i,j}$ the value of the i^{th} projection of the j^{th} basis function. Under those assumptions, the image reconstruction problem can be modeled as the inverse problem of estimating the x_j's from the y_i's by solving the system of linear equations aforementioned. Assuming that the whole set of equations in the linear system may be subdivided into *B* blocks, a generalized version of component averaging methods, *BICAV*, can be described. The processing of all the equations in one of the blocks produces a new estimate. All blocks are processed in one iteration of the algorithm. This technique produces iterates which converge to a weighted least squares solution of the system. A volume can be considered made up of 2D slices. The use of the spherically symmetric volume elements, blobs[10], makes slices interdependent because of blobs' overlapping nature. The main drawback of iterative methods are their high computational requirements. These demands can be faced by means of parallel computing and efficient reconstruction methods with fast convergence.

An adaptive parallel iterative reconstruction method is presented. The block iterative version of the component averaging methods, $BICAV$, has been parallelized following the Single Program Multiple Data (SPMD) approach[9]. The volume is decomposed into slabs of slices that will be distributed across the nodes (for MPI) or across the threads (for AMPI). Threads can, thereby, process their own data, however, the interdependence among neighbour slices due to the blob extension makes necessary the inclusion of redundant slices into the slabs. In addition, there must be a proper exchange of information between neighbour nodes. These communication points constitute synchronization points in every pass of the algorithm in which the nodes must wait for the neighbors. The amount of communications is proportional to the number of blocks and iterations. Reconstruction yields better results as the number of blocks is increased.

3 Latency Hiding with User Level Threads for BICAV

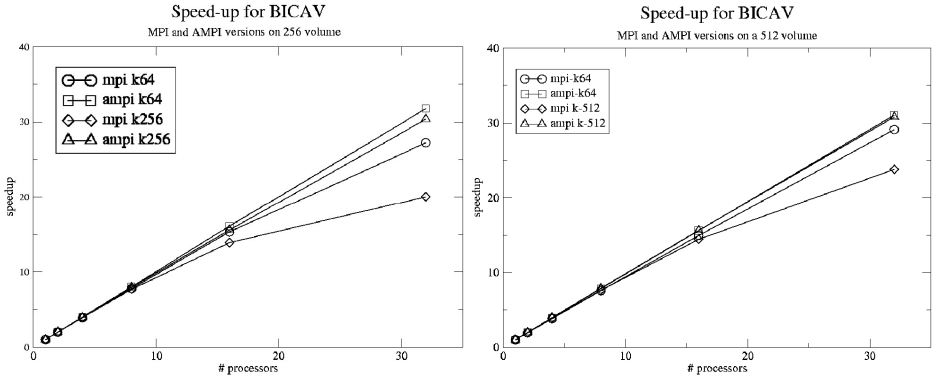
An analysis on the application's communication pattern has been carried out. Such analysis provides estimations on improvements to be achieved if the concurrence is really effective. Concurrence using AMPI user level threads offers the advantage of having more virtual processors than physical processors. Therefore more than one virtual processor can coexist in a physical processor efficiently. Tests carried out showed gains obtained by the multithreaded version of $BICAV$ compared to those obtained by the MPI version where latency hiding technique was included with non blocking Sends/Recvs. Scaling experiences were also performed for both versions, varying the number of threads/processor and the number of processors. It is important to note that for $BICAV$, as the number of blocks (*K*) increases, the convergence of the algorithm is faster, but the amount of communications also increases. This scenario harms the MPI version whereas AMPI is expected to keep good performance. Two volumes were used for testing $BICAV$, a 256 volume and a 512 volume.

All experiences were performed on *Vermeer*, our research cluster. This cluster runs a Gentoo Linux, Kernel version was 2.4.32 SMP. Each of its *32* computing nodes have two Pentium IV xeon 3.06 Ghz with 512 KB L2 Cache and 2 GB of ecc ddr sdram. Nodes were connected with 1 Gb Ethernet links.

Table 1. % Relative differences between CPU and WALL times for k 256 and k 512

	K 256 (volume 256)		K 512 (volume 512)	
	MPI	AMPI	MPI	AMPI
Procs	%	%	%	%
2	2.9	0.1	2.8	0.0
4	3.5	0	3.2	0.0
8	5.6	0	4.7	0.3
16	17.1	0.8	9.7	0.7
32	62.4	1.5	32.3	0.2

The relative differences between cpu and wall times are shown in Table 1, for both problem sizes, using the higher possible value for *K*, for each case. For AMPI, wall and cpu times are alike, which means that cpu was mostly in use, in contrast to the MPI version in which differences turn out to be significant. Taking into account that there are neither I/O operations nor function/method calls apart from those related to the algorithm itself, it can be said that for our multithreaded version, the concurrency is maximized. To further analyze the gains obtained by the multithreaded version of $BICAV$, the speedup was computed for the tests carried out for Table 1, see Fig. 1.



(a) 256 volume (b) 512 volume

Figure 1. Speedup in Vermeer cluster for 256 volume (left) and 512 volume (right)

In Fig. 1, wall times for MPI and AMPI (128 vps) versions are shown, for several numbers of blocks K and for 256 and 512 volume sizes. It can be observed that below a threshold of K=64 both versions seem to behave similarly, showing slight improvement in AMPI. But above that threshold, and as K increases, AMPI behaves better than MPI especially for more than *16* processors. $BICAV$'s AMPI version is getting benefits from the hidden concurrency. This means that the amount of communications needed due to a high value of *K* can be overlapped efficiently. These speedup curves show that with MPI, $BICAV$ loses the linear speedup when using more than eight processors. However, AMPI

keeps its speedup almost linear.

Once an optimal number of processors is found, an optimal number of threads per processor is also desirable, for both volumes. It was found that a number of threads ranging from four to eight were optimal. This means that having fewer threads than four, concurrency may not be exploited, on the other side, having a higher number than advised, performance can be degraded due to thread management overheads.

For non-dedicated clusters, concurrency, if exploited correctly, can play an important role for performance. Therefore, this criteria is implemented as a complement to the load balancing strategy. AMPI offers a threaded framework in which latencies due to communications can be handled. The following section shows how the load balancing strategy, RakeLB, takes advantage of latency hiding.

4 Adaptability

In general, load balancing strategies do not take into consideration data locality. Consequently, pieces of work sharing data can be placed on different processors. The strategy for preserving locality, RakeLB[5], was implemented as a centralized load balancer. For RakeLB, two processors are neighbours if each one has, at least, one thread with data in common. RakeLB has been implemented taking advantage of facilities provided by AMPI where the migration of workload among nodes is carried out in terms of threads. Employing user level threads make load balancing issues easier in SPMD applications. Each thread owns an universal ID, the rank. This rank establishes an order relationship between them. When computation is ignited, each thread picks up a chunk of data to work on, thereby threads with consecutive ranks will need, at some point in time to communicate one another. Therefore, thread migration decisions should conserve as much as possible the established thread relationship. To maintain the aforementioned order, a FIFO class structure has been devised. When threads migration is advised, those with the minimal data locality restrictions in the node will be migrated.

RakeLB behaviour has been contrasted with standard centralized, dynamic load balancing strategies, like Refine and Greedy[11]. Greedy strategy does not consider any previous thread-processor assignment, simply builds two queues, processors and threads, and reassigns load to reach average. Refine, in contrast, only migrates threads from overloaded processors until the processor load is pushed under average.

4.1 Implementation

The purpose of this load balancing algorithm is to redefine the initial threads distribution by applying a certain strategy, sensible to the criteria stated in Section 3. Inputs for strategy evaluation are provided by Charm (threads per processor, load per processor, cpu and wall time per thread,. . .). For preserving data locality, RakeLB has to determine the set of available processors in the cluster to create logical links between processors that host *correlative* sets of threads (according to the established order relationship). RakeLB determines, for each processor, its previous and next available processors, from a logical point of view. A procedure called $LinkProcs$ has been developed to provide such a facility. $LinkProcs$ is responsible of maintaining processors virtually linked, providing therefore, a consistent way to prevent migrations between sets of non correlative threads. This procedure also

Algorithm 3 : RakeLB

```
1.  for i = 1 to numAvail − Resd                    First phase: numAvail − Resd iterations
2.    for j = 1 to NP                                          Compute total load as
3.        Burd_j = PRCS_j.bkgLoad + PRCS_j.cmpLoad;            bg load + threads' load
4.    endfor
5.    for j = 1 to NP
6.      while (Burd_j > AvLoad & numThreads_j > 0)             while proc is overloaded
7.        deAssign(THRD.id, PRCS_j);                  move away threads, preserving locality
8.        Assign(THRD.id, PRCS_j.next);                          to next processor
9.        numThreads_j = numThreads_j − 1;                       update #threads
10.       Burd_j = Burd_j − THRD.cmpLoad;                        update load
11.     endwhile
12.   endfor
13. endfor
14. for i = 1 to Resd                                  Second phase: Resd iterations
15.   for j = 1 to NP
16.       Burd_j = PRCS_j.bkgLoad + PRCS_j.cmpLoad;
17.   endfor
18.   for j = 1 to NP
19.     while (Burd_j > AvLoad + ΔLoad &numThreads_j > 0)    note ΔLoad
20.       deAssign(THRD.id, PRCS_j);
21.       Assign(THRD.id, PRCS_j.next);
22.       numThreads_j = numThreads_j − 1;
23.       Burd_j = Burd_j − THRD.cmpLoad;
24.     endwhile                  During migrations, Assign and Deassign methods
25.   endfor                      together with LinkProcs and the FIFO structure
26. endfor                        devised, preserved data locality for threads
```

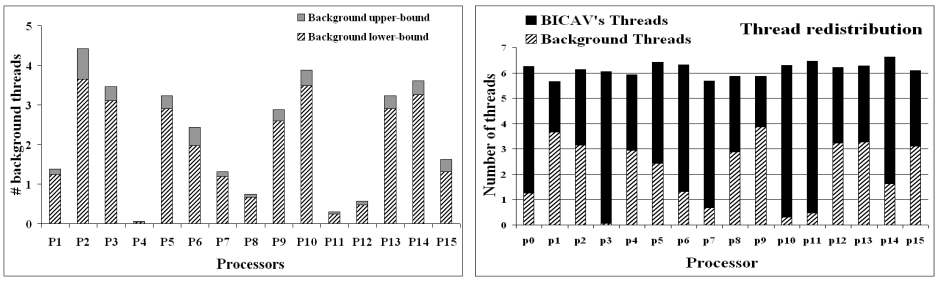
maintains useful information such as the number of available processors ($numAvail$), the average load ($AvLoad$) which is worked out accumulating all the load in each available processor. Hence $AvLoad$ is the main parameter concerning thread migration, together with $Resd$, which is the remainder obtained when calculating $AvLoad$. Another important parameter is the total number of $BICAV$ threads, $NThreads$.

RakeLB convergence is assured in $numAvail$ iterations. These $numAvail$ iterations are performed in two main steps. The first step consists of $numAvail - Resd$ iterations, the second step consists of *Resd* iterations. At the first step, RakeLB examines processors whose load is over the average ($AvLoad$) and determines which threads migrate to *next* processor. For every iteration, the total load in each processor is re-computed and stored in the variable $Burd$. There is an upper-bound limit, $numThreads$, to the number of threads that can be migrated from an overloaded processor. Note that the total number of threads in the application is given by $NThreads = \sum_{i=1}^{numAvail} numThreads_i$. At the second step only those processors whose load is over $AvLoad$+Δ $Load$ are involved. Δ $Load$ is set, during these experiences, to the minimum thread load.

5 Evaluation of Load Balancing Strategies

Preserving data locality and minimizing latencies are two issues that the RakeLB strategy exploits. In this section, a study of RakeLB, GreedyLB and RefineLB (Non-Locality Load Balancing Algorithms) behaviour is presented. GreedyLB and RefineLB are implementations for Greedy and Refiner strategies.

$BICAV$ is used to evaluate the dynamic load balancing algorithms. $BICAV$ exhibits strong data dependence and hence emphasizes the influence of data locality preservation on the performance. Load balancer will be invoked just once.



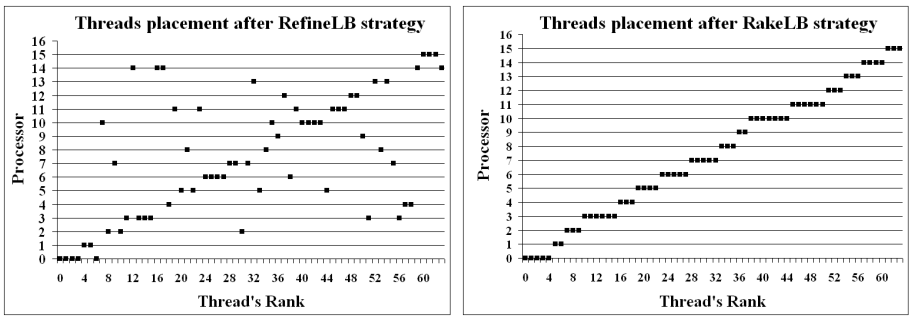
(a) Background (max and min values) per processor. (b) Threads after RefineLB and RakeLB strategies

Figure 2. Background load and resulting data redistribution after load balancing.

Background load scenarios were prepared using threads from a secondary application, in order to create a controlled imbalance. Fig. 2(a) exposes the number of background threads placed per processor. The background load evolution simulates other user's interactions with the cluster. For every experiment, the maximum value for the background activity is controlled by a parameter, $maxback$, adapted to the current number of $BICAV$'s threads per processor, that is, $NThreads \div numAvail$. Background load is assigned randomly to a processor, having a minimum value of zero and a maximum value of $maxback$, see Fig. 2(a). At the beginning, computing threads for the application are evenly distributed among nodes.

Fig. 2 shows how the strategies under study react to load imbalanced scenarios. As discussed in Section 3, from sixteen processors $BICAV$ show significant divergences between the AMPI version and the MPI version. A number ranging from *4* to *8* vps per processor was advised. Thus the sixteen processors and sixty four threads ($NThreads = 64$) case is selected for testing load balancing strategies.

After migration, see Fig. 2(b), RakeLB and RefineLB reacted alike. The decision that GreedyLB took follows the trend established by RakeLB and RefineLB, although its distribution diverges slightly from that taken by its counterpart strategies. What can be deduced from these figures is that all three strategies react in a very similar way to the effect of the background load. There's a key point, viz. *data dependency*. An analytical proof for what is asserted in Fig. 2 was performed employing the standard deviation of the whole system load, normalized to the average load, σ. The initial imbalance value reflected by σ was over 0.5. After applying the balancing strategies a similar σ value (0.051 for GreedyLB



(a) Disorder after RefineLB (b) Locality preservation after RakeLB

Figure 3. Placement of threads

Table 2. Walltime ratio and overlap ratio for $BICAV$

	Greedy	Refine	Rake
Cputime / Walltime ratio	0.34	0.52	0.70
Master's Walltime (% relative to Greedy)	100	66.59	50.29

and 0.045 for RakeLB and RefineLB) was achieved, but there's a key point *data dependency*.

In Fig. 2(b), it can be seen that after the load balancer is done, a complete homogeneous load distribution is achieved (σ almost zero). Although load balancers strategies achieve an almost equal load distribution (alike for RakeLB and Refine) it is important to note that these distributions are completely different in terms of data locality, see Fig. 3, aspect that turns out to be an issue for $BICAV$'s performance. Only RefineLB and RakeLB figures are shown to state that although both σ values are similar, data distributions are different. GreedyLB distribution was remarkably messy.

Maintaining the advisable number of neighboring threads in the same processor as RakeLB does, leads to a performance improvement in the application. This performance improvement begins to be much more significant for $BICAV$ when balanced with the RakeLB strategy in contrast with the cases where it is balanced with Refine or Greedy strategies, from the case where sixteen processors are used, see Table 2. This is the reason why in general terms, with RakeLB, $BICAV$, is faster on *vermeer* cluster. This is found in the combination of both issues under study. With RakeLB, latencies are lower than with any of the other two strategies, having enough computation to overlap communication, which is not the case with RefineLB and GreedyLB. This overlap is seized by threads sharing the same processor to advance computation.

6 Conclusions

Two concepts, adaptability and latency hiding abilities, were studied together, holding the hypothesis that for scientific applications where data is under a tight data locality rela-

tionship, both could be combined in such a way that executions on non dedicated clusters could obtain good results indeed in contrast to not considering these concepts. On the one hand, the use of user level threads allowed multiple flows of control in one processor, this characteristic together with their fast context switching operation times, makes it very easy to hide communication operations with other thread computations. On the other hand, load balancing with strategies that maintains a thread's neighbourhood -regarding data locality- avoids overloaded processors that harm the performance of the applications. If such strategies, as shown, maintain data locality, then latencies to be hidden are reduced, having as only handicap the possible overhead caused by the thread's management. The dynamic load balancing strategy which preserves data locality, was proved to be efficient for applications like $BICAV$.

Acknowledgements

This work has been supported by grants MEC-TIN2005-00447 and JA-P06-TIC01426.

References

1. C. M. Pancake, *Multithreaded Languages for Scientific and Technical Computing*, Proceedings of the IEEE, **81**, 2, (1993).
2. S. Oikawa, H. Tokuda, *Efficient Timing Management for User-Level Real-Time Threads*, in: Proc. IEEE Real-Time Technology and Applications Symposium, 27–32, (1995).
3. G. W. Price, D. K. Lowenthal, *A comparative analysis of fine-grain threads packages*, Parallel and Distributed Computing, **63**, 1050–1063, (2003) College Station, Texas.
4. A. C. Kak, M. Slaney, *Principles of Computerized Tomographic Imaging*, (SIAM Society for Industrial and Applied Mathematics, 2001).
5. C. Fonlupt, P. Marquet, J. L. Dekeyser, *Data-parallel load balancing strategies*, Parallel Computing, **24**, 1665-1684, (1998).
6. Ch. Huang, O. Lawlor, L. V. Kale, *Adaptive MPI*, in: Proc. 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03), College Station, Texas, 306–322, (2003).
7. Ch. Huang, G. Zheng, S. Kumar, L. V. Kale, *Performance evaluation of Adaptive MPI* in: Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (2006).
8. L. V. Kale, S. Krishnan, *Charm++: a portable concurrent object oriented system based on C++*, in: Proc. Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM Press, 91–108, (1993).
9. J. Fernández, A. F. Lawrence, J. Roca, I. García, M. H. Ellisman, J. M. Carazo, *High performance electron tomography of complex biological specimens*, Journal of Structural Biology, **138**, 6–20, (2002).
10. S. Matej, R. Lewitt, G. Herman, *Practical considerations for 3-D image reconstruction using spherically symmetric volume elements*, IEEE Trans. Med. Imag., **15**, 68–78, (1996).
11. G. Aggarwal, R. Motwani, Zhu, *The load rebalancing problem*, Journal Algorithms, **60**, 42–59, (2006).